# Software Architecture Document

## <Project Name>

## Revisions

|  |  |  |
| --- | --- | --- |
| Version | Date | Author |
| 1.0 | 08/01/2017 | S. Rossokha |
|  |  |  |
|  |  |  |

## Context

A context section should answer the following types of questions:

• What is this software project/product/system all about?  
• What is it that’s being built?  
• How does it fit into the existing environment? (e.g. systems, business processes, etc) • Who is using it? (users, roles, actors, personas, etc)

## Functional Overview

This section allows you to summarize what the key functions of the system are. It also allows you to make an explicit link between the functional aspects of the system (use cases, user stories, etc) and, if they are significant to the architecture, to explain why. A functional overview should answer the following types of questions:

* + Is it clear what the system actually does?
  + Is it clear which features, functions, use cases, user stories, etc are significant to the   
    architecture and why?
  + Is it clear who the important users are(roles, actors, personas, etc)and how the system   
    caters for their needs?
  + It is clear that the above has been used to shape and define the architecture?

Alternatively, if your software automates a business process or workflow, a functional view should answer questions like the following:

* + Is it clear what the system does from a process perspective?
  + What are the major processes and flows of information through the system?

## Quality Attributes

This section is about summarizing the key quality attributes and should answer the following types of questions:

* Is there a clear understanding of the quality attributes that the architecture must satisfy?
* Are the quality attributes SMART (specific, measurable, achievable, relevant and timely)?
* Havequalityattributesthatareusuallytakenforgrantedbeenexplicitlymarkedasout of scope if they are not needed? (e.g. “user interface elements will only be presented in English” to indicate that multi-language support is not explicitly catered for)   
  Are any of the quality attributes unrealistic? (e.g. true 24x7 availability is typically very costly to implement inside many organizations)

In addition, if any of the quality attributes are deemed as “architecturally significant” and therefore influence the architecture, why not make a note of them so that you can refer back to them later in the document.

Simply listing out each of the quality attributes is a good starting point. Examples include:   
• Performance (e.g. latency and throughput)  
• Scalability (e.g. data and traffic volumes)  
• Availability (e.g. uptime, downtime, scheduled maintenance, 24x7, 99.9%, etc)

• Security (e.g. authentication, authorization, data confidentiality, etc)

• Extensibility  
• Flexibility  
• Auditing  
• Monitoring and management • Reliability

* Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)
* Business continuity
* Interoperability
* Legal, compliance and regulatory requirements (e.g. data protection act)
* Internationalisation (i18n) and localisation (L10n)
* Accessibility
* Usability
* ...

## Constraints

Constraints are typically imposed upon you but they aren’t necessarily “bad”, as reducing the number of available options often makes your job designing software easier. This section allows you to explicitly summarize the constraints that you’re working within and the decisions that have already been made for you.

As with the quality attributes, simply listing the known constraints and briefly summarizing them will work. Example constraints include:

• Time, budget and resources.  
• Approved technology lists and technology constraints.  
• Target deployment platform.  
• Existing systems and integration standards.  
• Local standards (e.g. development, coding, etc).  
• Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc). • Standard protocols.  
• Standard message formats.  
• Size of the software development team.  
• Skill profile of the software development team.  
• Nature of the software being built (e.g. tactical or strategic).  
• Political constraints.  
• Use of internal intellectual property.  
• etc

## Principles

The purpose of this section is to simply make it explicit which principles you are following. These could have been explicitly asked for by a stakeholder or they could be principles that you (i.e. the software development team) want to adopt and follow.

If you have an existing set of software development principles (e.g. on a development wiki), by all means simply reference it. Otherwise, list out the principles that you are following and accompany each with a short explanation or link to further information. Example principles include:

* + Architectural layering strategy.
  + No business logic in views.
  + No database access in views.
  + Use of interfaces.
  + Always use an ORM.
  + Dependency injection.
  + The Hollywood principle (don’t call us, we’ll call you).
  + High cohesion, low coupling.
  + Follow SOLID (Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle).
  + DRY (don’t repeat yourself).
  + Ensure all components are stateless (e.g. to ease scaling).
  + Prefer a rich domain model.
  + Prefer an anaemic domain model.
  + Always prefer stored procedures.
  + Never use stored procedures.
  + Don’t reinvent the wheel.
  + Common approaches for error handling, logging, etc.
  + Buy rather than build.
  + etc

## Software Architecture

The purpose of this section is to summarize the software architecture of your software system so that the following questions can be answered:

* + What does the “big picture” look like?
  + Is there are clear structure?
  + Is it clear how the system works from the “30,000 foot view”?
  + Does it show the major containers and technology choices?
  + Does it show the major components and their interactions?
  + What are the key internal interfaces? (e.g. a web service between your web and   
    business tiers)

Use the **C4** container and component diagrams from as the main focus for this section, accompanied by a short narrative explaining what the diagram is showing plus a summary of each container/component.

Sometimes UML sequence or collaboration diagrams showing component interactions can be a useful way to illustrate how the software satisfies the major use cases/user stories/etc.

## External Interfaces

The purpose of this section is to answer the following types of questions:

• What are the key external interfaces?

–  e.g. between your system and other systems (whether they are internal or   
external to your environment)

–  e.g. any APIs that you are exposing for consumption

–  e.g. any files that your are exporting from your system   
• Has each interface been thought about from a technical perspective?

–  What is the technical definition of the interface?

–  If messaging is being used, which queues (point-to-point) and topics (pub-sub)   
are components using to communicate?

–  What format are the messages (e.g. plain text or XML defined by a DT-   
D/Schema)?

–  Are they synchronous or asynchronous?

–  Are asynchronous messaging links guaranteed?

–  Are subscribers durable where necessary?

–  Can messages be received out of order and is this a problem?

–  Are interfaces idempotent?

–  Is the interface always available or do you, for example, need to cache data   
locally?

–  How is performance/scalability/security/etc catered for?   
• Has each interface been thought about from a non-technical perspective? – Who has ownership of the interface?  
 – How often does the interface change and how is versioning handled? – Are there any service-level agreements in place?

## Code

The purpose of the code section is to describe the implementation details for parts of the software system that are important, complex, significant, etc. For example, I’ve written about the following for software projects that I’ve been involved in:

* + Generating/rendering HTML: a short description of an in-house framework that was created for generating HTML, including the major classes and concepts.
  + Data binding: our approach to updating business objects as the result of HTTP POST requests.
  + Multi-page data collection: a short description of an in-house framework we used for building forms that spanned multiple web pages.
  + Web MVC: an example usage of the web MVC framework that was being used.
  + Security: our approach to using Windows Identity Foundation(WIF) for authentication   
    and authorization.   
    Domain model: an overview of the important parts of the domain model.
  + Component framework: a short description of the framework that we built to allow   
    components to be reconfigured at runtime.
  + Configuration: a short description of the standard component configuration mechanism in use across the codebase.
  + Architectural layering: an overview of the layering strategy and the patterns in use to   
    implement it.
  + Exception sand logging: a summary of our approach to exception handling and logging across the various architectural layers.
  + Patterns and principles: an explanation of how patterns and principles are implemented.
  + etc

## Data

The purpose of the data section is to record anything that is important from a data perspective, answering the following types of questions:

* + What does the data model look like?
  + Where is data stored?
  + Who owns the data?
  + How much storage space is needed for the data? (e.g. especially if you’re dealing with   
    “big data”)
  + What are the archiving and back-up strategies?
  + Are there any regulatory requirements for the long term archival of business data?
  + Likewise for log files and audit trails?
  + Are flat files being used for storage? If so, what format is being used?

## Infrastructure Architecture

This section is used to describe the physical/virtual hardware and networks on which the software will be deployed. Although, as a software architect, you may not be involved in designing the infrastructure, you do need to understand that it’s sufficient to enable you to satisfy your goals. The purpose of this section is to answer the following types of questions:

* + Is there a clear physical architecture?
  + What hardware (virtual or physical) does this include across all tiers?
  + Does it cater for redundancy, failover and disaster recovery if applicable?
  + Is it clear how the chosen hardware components have been sized and selected?
  + If multiple servers and sites are used, what are the network links between them?
  + Who is responsible for support and maintenance of the infrastructure?
  + Are there central teams to look after common infrastructure (e.g. databases, message buses, application servers, networks, routers, switches, load balancers, reverse proxies, internet connections, etc)?
  + Who owns the resources?
  + Are there sufficient environments for development, testing, acceptance, pre-production, production, etc?

The main focus for this section is usually an infrastructure/network diagram showing the various hardware/network components and how they fit together, with a short narrative to accompany the diagram.

## Deployment

This section is used to describe the mapping between the software (e.g. containers) and the infrastructure. Sometimes this will be a simple one-to-one mapping (e.g. deploy a web application to a single web server) and at other times it will be more complex (e.g. deploy a web application across a number of servers in a server farm). This section answers the following types of questions:

* + How and where is the software installed and configured?
  + Is it clear how the software will be deployed across the infrastructure elements described in the infrastructure architecture section? (e.g. one-to-one mapping, multiple containers per server, etc)
  + If this is still to be decided, what are the options and have they been documented?
  + Is it understood how memory and CPU will be partitioned between the processes running on a single piece of infrastructure?
  + Are any containers and/or components running in an active-active, active-passive, hot-standby, cold-standby, etc formation?
  + Has the deployment and rollback strategy been defined?
  + What happens in the event of a software or infrastructure failure?
  + Is it clear how data is replicated across sites?

There are a few ways to structure this section:

* 1. Tables: simple textual tables that show the mapping between software containers and/or components with the infrastructure they will be deployed on.
  2. Diagrams: UML deployment diagrams or modified versions of the diagrams from the infrastructure architecture section showing where software will be running.

## Operation and Support

Most systems will be subject to support and operational requirements, particularly around how they are monitored, managed and administered. Including a dedicated section in the software guidebook lets you be explicit about how your software will or does support those requirements. This section should address the following types of questions:

* 1. Is it clear how the software provides the ability for operation/support teams to monitor and manage the system?
  2. How is this achieved across all tiers of the architecture?
  3. How can operational staff diagnose problems?
  4. Where are errors and information logged? (e.g. log files, Windows Event Log, SMNP,
  5. JMX, WMI, custom diagnostics, etc)
  6. Do configuration changes require a restart?
  7. Are there any manual house keeping tasks that need to be performed on a regular basis?
  8. Does old data need to be periodically archived?

## Development Environment

The purpose of this section is to provide instructions that take somebody from a blank operating system installation to a fully-fledged development environment.

The type of things you might want to include are:

* + Pre-requisite versions of software needed.
  + Links to software downloads (either on the Internet or locally stored).
  + Links to virtual machine images.
  + Environment variables, Windows registry settings, etc.
  + Host name entries.
  + IDE configuration.
  + Build and test instructions.
  + Database population scripts.
  + Usernames, passwords and certificates for connecting to development and test services.
  + Links to build servers.
* etc

If you’re using automated solutions (such as Vagrant, Docker, Puppet, Chef, Rundeck, etc), it’s still worth including some brief information about how these solutions work, where to find the scripts and how to run them.

## Decision Log

The purpose of this section is to simply record the major decisions that have been made, including both the technology choices (e.g. products, frameworks, etc) and the overall architecture (e.g. the structure of the software, architectural style, decomposition, patterns, etc).

For example:

* Why did you choose technology or framework “X” over “Y” and “Z”?
* How did you do this? Product evaluation or proof of concept?
* Were you forced into making a decision about “X” based upon corporate policy or enterprise architecture strategies?
* Why did you choose the selected software architecture? What other options did you consider?
* How do you know that the solution satisfies the major non-functional requirements?
* etc